Minimizing the Overhead of Effect Systems Using Capabilities

Anlun Xu
Carnegie Mellon University
Pittsburgh, PA
anlunx@andrew.cmu.edu

ABSTRACT

Effect systems are a promising approach for reasoning about the side effects of code. This research study focused on extending the effect system based on capabilities in Wyvern programming language, to minimize its overhead. Specifically, the research study implements the inference of import bounds and evaluates its impact on the usability of Wyvern programing language.

KEYWORDS

Effect System, Effect Polymorphism, Capabilities, Higher Order Functions

1 Introduction

1.1 Background

Effect systems allow programmers to reason about the side effect of the code, such as reads and writes to memory, exceptions, and I/O operations [3]. Java's checked exceptions is a simple effect system that is widely used. Effect systems could be used to ensure that any untrusted code can only access the system resource that programmers allow it to get access to.

The effect system in Wyvern programming language allows programmers to reason about the side effect of Wyvern code. The effect system in Wyvern allows programmers to define abstract effects in types to hide the lower level definitions of effects. For example. In the Wyvern standard library, we define type Writer as

```
resource type Writer
   effect write
   effect close
   def write(s: String): {this.write} Unit
   def close(): {this.close} Unit
```

Figure 1: Definition of type Writer

This is a type that describes a writer with two effects: write and close. A module that has writer type will have four members: effect write and effect close are two effect members of Writer. While write and close are two member functions. Now we consider a possible module of this type:

```
val appender : Writer = new
       effect write = {system.FFI}
       effect close = {system.FFI}
       def write(s : String) : {self.write} Unit
              // Calling foreign function interface
       def close : {this.close} Unit
              // Calling foreign function interface
```

Figure 2: Definition of appender

We can see that both write effect and close effect are defined as {system.FFI}, which represents the effect of calling a foreign function interface. Therefore, the effect system in Wyvern supports effect abstraction, which allows programmers to reason about the side effect of a program based on higher-level abstractions of effects, and hide the lower-level definition of effects. In this example, the {system.FFI} is hidden from the user of type Writer, and the user of type Writer can analyze the write effect and close effect separately even though they could have the same definition.

Moreover, the Wyvern programing language is a capability-safe language, meaning that there is no global state or globally accessible modules that provide access to system resources [3]. Therefore, a module cannot have any side effect if there is no other resource module that is given to that module. In Figure 2, we have defined a module that can cause write effect and close effect. If we pass the appender module into another module, the other module will have the ability to cause write effect or close effect. Otherwise, the other module will never have access to these two effects.

Therefore, the effect system, and the capability safe module system in Wyvern programming language allows programmers to analyze the effect in a module, and restrict the system resources that one module can have access to.

1.2 Motivation

Although the effect system in Wyvern programming language allows the programmers to analyze the effect of the code. The downside of the system is that it creates a large overhead for programmers in terms of the number of effect annotations. For example, a library that uses an object writer with type Writer to perform operations such as write or close would have to annotate its functions with {writer.write} and {writer.close}.

```
module def fileAppender(writer : Writer) : {}
  def appendFile(s : String) : {writer.write, writer.close} Unit
    writer.write(s)
    writer.close()
```

Figure 3: Effect annotated fileAppender

fileApender is a module with access to a module of type Writer. Since member functions in writer are annotated with effects, we need to annotate appendFile with the write and close effect. This can be problematic since we need to annotate every function in a module in order to make the program fully effect-checked. Therefore, we want to find a way to analyze the effect in a module without annotating every function in the module.

1.3 Approach

Since Wyvern modules are capability-safe, we use capabilities to reason about effects in modules. We demonstrate that the capability provides us with a way to bound the effects of expressions in Wyvern. The reasoning on effect bound only happens in type-checking and does not require the programmer to add effect annotations inside the expression, and does not require the expression to be analyzed for its effects.

1.4 Related Work

This effect bound inference for Wyvern programs in this research is based on the result of Craig et al. In their paper, *Capabilities-Effect for Free,* Craig et al proposed a way to compute the effect bound on expressions in a calculus [2]. The capability-safe module system and its impact on the effect system are examined by Melicher et al. in their paper Effect Abstraction: Applications to Security [3]. The implementation provided by this research study is based on the work by Justin Lubin on polymorphic effect approximation [1].

1.5 Contributions

This research study provides a working effect system in a full-fledged programming language that allows programmers to use the effect-unannotated code in effect-annotated code safely. We have also conducted a case study to show that the new effect system in Wyvern reduces the number of effect annotation needed in programs and therefore increase the usability of the Wyvern programming language.

2 Design and Approach

2.1 Module Lifting

```
module def fileAppender(writer : Writer)
  def appendFile(s : String) :Unit
    writer.write(s)
    writer.close()
```

Figure 4: Effect unannotated fileAppender

We use the keyword 'lifted' to indicate that we want to select the effect bound of an effect-unannotated module. For example, if we have an unannotated version of the module fileAppender (Figure 4), and want to use this module safely. We need to tell the compiler that we lift the effect of this module out and insert the effect bound when we call the functor in this module.

```
import lifted fileAppender
val fileappender = fileAppender[{appender.write, appender.close}](appender)
```

Figure 5: Using an unannotated library

We use the effect set {appender.write, appender.close} as the effect bound of this module, because appender has type Writer, and a module with type Writer can cause write effect and close effect.

When the first line of the code in Figure 5 is compiled, the lifted module will be treated by the compiler as the following:

```
module def fileAppender[effect E](writer : Writer)
  def appendFile(s : String) : {E} Unit
    writer.write(s)
    writer.close()
```

Figure 6: Lifted fileAppender

This transformation ensures that the effect E is the effect of any function inside the module fileAppender.

2.2 Effect Separation

We separate the effect-annotated code and effect-unannotated code to ensure the safety of a program. That is to say if we want a module to be fully effect-checked. We will ensure that the module only depends on effect-annotated modules or lifted effect-unannotated modules. We use effect annotation on module definitions to indicate that the module is fully effect-checked.

```
module def fileAppender(writer : Writer) : {}
  def appendFile(s : String) :Unit
    writer.write(s)
    writer.close()
```

Figure 7: Effect-checked module with empty effect

For example, in figure 7 we defined a fileAppender with empty effect annotation. The effect annotation indicates that the module is fully effect checked. However, the appendFile function inside the module is not effect-annotated. Therefore, the compiler will complain and notify the programmer to annotate the effects of appendFile function.

2.3 Effect Bound Inference

$$\frac{e : \tau_1 \to \tau_2 \quad L = effects(\tau_1) \cup \text{hoeffects}(\tau_1) \quad U = \{\epsilon \mid hosafe(\tau_1, \epsilon)\}}{lifted\ e : \forall \epsilon (L \subset \epsilon \subset U).\tau_1 \to \tau_2(\epsilon)}$$

Figure 8: Inference rule for computing the effect bound

The effect bound on a Wyvern module is computed based on the capabilities that are passed into the module. The main rule we used when computing the effect bound on a module depends on the rules proposed by Carig et al [2]. The inference rule in Figure 8 computes both the upper bound and the lower bound of the selected effect of a module functor when there are higher order functions in capabilities that are passed into the lifted module.

```
val plugin = new
  effect write = {system.FFI}
  def fun() : {} (Unit -> {this.write} Unit)
    () => Unit
```

Figure 9: Definition of plugin capability

To understand how the lower bound on effects of unannotated modules is computed, consider a lifted functor named pluginUser that receives a resource module named plugin as an argument. We define plugin in Figure 9.

Although the plugin itself does not contain a member function that causes an effect, we can see that the plugin contains a function of type {} => (Unit -> {this.write} Unit), which creates a function that causes a write effect when provided with a unit value. Therefore, we need to add the write effect into the effect lower bound of the module that is created by calling pluginUser with plugin as its argument.

```
val pluginuser1 = pluginUser[{}](plugin)
val pluginuser2 = pluginUser[{plugin.write}](plugin)
```

Figure 10: Effect bound on pluginUser module

In Figure 10, the two lines of code call the pluginUser functor and instantiate two modules, pluginuser1 and pluginuser2. According to our effect bound inference, the first line will be rejected by the compiler since the plugin.write effect is in the lower bound of the module, but is not selected by the functor call. However, the second line of code will be accepted by the compiler because the selected effect contains the lower bound of the module, and is effect-safe.

Moreover, we also compute the upper bound on the effect of modules. Consider two resource modules go and file

```
val go = new
  def run(f : Unit -> {} Unit) : {} Unit
    f()

val file = new
  effect write = {system.FFI}
  def writeFile() : {this.write} Unit
    // Calling foreign function interface
```

Figure 11: Definitions of go and file

go is a module with a member function run, which receives an effect-free function and executes it. file is a module with the write effect and a function that causes write effect. If we pass these two capabilities into an unannotated module, the unannotated module is allowed to pass file.writeFile into go.run. However, we should not allow this since the function argument of go.run should be effect-free. Therefore, we set the upper bound on the effect of the unannotated module to be the empty effect. Consider calling an unannotated module functor fileUser:

```
val fileuser = fileUser[{file.write}](go, file)
```

Figure 12: A functor using go and file

This line of code would be rejected by the compiler since {file.write} exceeds the upper bound of fileUser, which is the empty effect.

3 Evaluation of the Approach

3.1 Case study

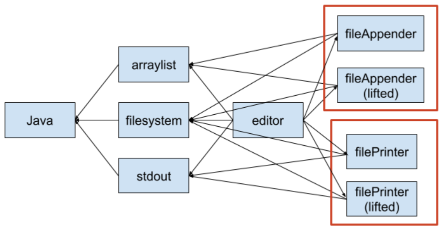


Figure 13: Editor Case Study

To evaluate the effect of the effect system on the usability of Wyvern programming language. We construct an editor application that depends on system resource modules such as arraylist, filesystem, and stdout. These modules are part of the Wyvern standard library and are effect-annotated in order to be used in the case study. The editor application also uses on two library modules, fileAppender, and filePrinter, that depend on system resources and have effects. We also create unannotated versions of fileAppender and filePrinter, and import the lifted versions of these two modules.

The purpose of this case study is to compare the impact of importing lifted modules on the number of effect annotations, and therefore evaluate the impact of the effect system on the usability of Wyvern programming language.

3.2 Result

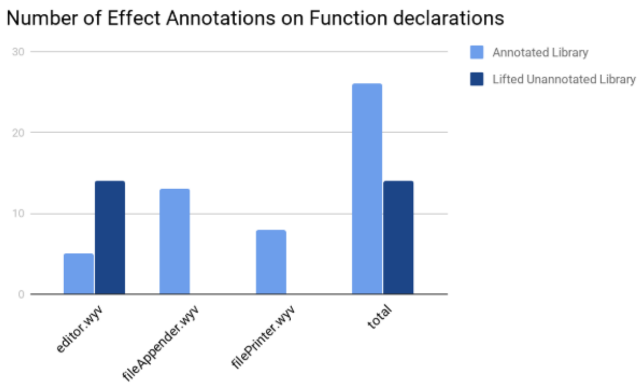
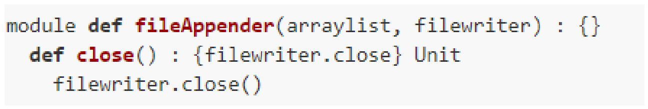


Figure 14: Number of Effect Annotations on Function Declaration

We compare the number of effect annotations in the editor function between the case when the two libraries are fully annotated and the case when the two libraries are not annotated but lifted when imported. We can observe that the number of annotations required increased in the editor module if we lift the libraries because the effect bounds on the libraries are computed conservatively. However, since we don't need to add annotations in lifted fileAppender and lifted filePrinter, the total number of effect annotations decreased.

To see why the number of effect annotations increased in editor.wyv when we lift the library modules, consider an annotated module fileAppender with a member function close:



```
module def fileAppender(arraylist, filewriter) : {}
  def close() : {filewriter.close} Unit
    filewriter.close()
```

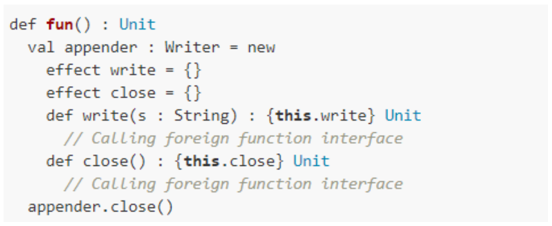
Figure 15: Annotated fileAppender module

If we annotate the effect of close in editor.wyv, we only need to write {filewriter.close} as its effect. However, if fileAppender is unannotated, the client does not know what effect the close function has and has to write the union of effect sets of functions in arraylist and filewriter as the effect annotation for the function close. So, the number of annotations can increase if we use the lifted unannotated library instead of the annotated library.

We also observed that the effect system can be more effective when the library module contains a lot of functions that cause effects. Since by lifting the module, we get rid of every effect annotation on member functions, we achieve a better result if the library module is more complex and contain more functions. In Figure 14, we notice that fileAppender contributed more decrease in effect annotations because the module is more complex than the filePrinter module.

4 Limitations and Future Work

The biggest limitation of the effect system in Wyvern is that we cannot express the effect of the function when a resource module that is instantiated inside of the function because the effect of the instantiated module is not in scope when we define the function. To see this, consider the function fun



```
def fun() : Unit
  val appender : Writer = new
    effect write = {}
    effect close = {}
    def write(s : String) : {this.write} Unit
      // Calling foreign function interface
    def close() : {this.close} Unit
      // Calling foreign function interface
  appender.close()
```

Figure 16: Function with local module

The effect annotation for fun should be {appender.close}. However, we cannot write this effect annotation since appender is not in scope when we declare the function fun.

In the future, we can implement a hierarchy of effects in order to express the effect in modules which are not yet instantiated.

5 Conclusion

This study designs and implements a system that allows programmers to safely use the effect-unannotated code in the effect-annotated code. Specifically, we implemented the module lifting mechanism to allow programmers to select the effect of unannotated modules. Moreover, we implemented the inference of import bounds using the capability-based module system in Wyvern, which allows the unannotated code to be safely combined with annotated code. Finally, this study shows that lifting the effect of modules helps alleviating the overhead of effect annotations and increase the usability of the effect system in general.

ACKNOWLEDGEMENTS

This project was supervised by Jonathan Aldrich at Carnegie Mellon University (USA) and Alex Potanin at Victoria University of Wellington (NZ).

REFERENCES

[1] JustinLubin.2018.ApproximatingPolymorphicEffectswithCapabilities.In Proceedings of ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH'18). ACM, New York, NY, USA, 3 pages. https://doi.org/10.475/123_4

[2] Aaron Craig, Alex Potanin, Grove Lindsay and Jonathan Aldrich, Capabilities: Effects for Free, Formal Methods and Software Engineering

[3] Darya Melicher, 2018 Effect Abstraction: Application to Security

[4] JosephR.Kiniry.2006. Exceptions in Java and Eiffel: Two Extremes in Exception Design and Application. Springer Berlin Heidelberg, Berlin, Heidelberg,288–300. https://doi.org/10.1007/11818502_16

[5] Dominique Devriese, Lars Birkedal, and Frank Piessens. 2016. Reasoning about Object Capabilities with Logical Relations and Effect Parametricity.In2016 IEEE European Symposium on Security and Privacy (EuroS P).147–162. https://doi.org/10.1109/EuroSP.2016.22

[6] Ligia Nistor, Darya Kurilova, Stephanie Balzer, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. 2013. Wyvern: A Simple, Typed, and Pure Object-oriented Language.In Proceedings of the 5th Workshop on Mech Anisms for SPEcialization, Generalization and in HerItance (MASPEGHI '13). ACM, New York, NY, USA, 9–16. https: //doi.org/10.1145/2489828.2489830

[7] Valerie Zhao.2017. Abstracting Resource Effects. In Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion 2017).ACM,NewYork,NY,USA,48–50. https://doi.org/10.1145/3135932.3135946